

55. IWK

Internationales Wissenschaftliches Kolloquium
International Scientific Colloquium



13 - 17 September 2010

Crossing Borders within the **ABC**

Automation,

Biomedical Engineering and

Computer Science



Faculty of
Computer Science and Automation

www.tu-ilmenau.de

th
TECHNISCHE UNIVERSITÄT
ILMENAU

Home / Index:

<http://www.db-thueringen.de/servlets/DocumentServlet?id=16739>

Impressum Published by

Publisher: Rector of the Ilmenau University of Technology
Univ.-Prof. Dr. rer. nat. habil. Dr. h. c. Prof. h. c. Peter Scharff

Editor: Marketing Department (Phone: +49 3677 69-2520)
Andrea Schneider (conferences@tu-ilmenau.de)

Faculty of Computer Science and Automation
(Phone: +49 3677 69-2860)
Univ.-Prof. Dr.-Ing. habil. Jens Haueisen

Editorial Deadline: 20. August 2010

Implementation: Ilmenau University of Technology
Felix Böckelmann
Philipp Schmidt

USB-Flash-Version.

Publishing House: Verlag ISLE, Betriebsstätte des ISLE e.V.
Werner-von-Siemens-Str. 16
98693 Ilmenau

Production: CDA Datenträger Albrechts GmbH, 98529 Suhl/Albrechts

Order trough: Marketing Department (+49 3677 69-2520)
Andrea Schneider (conferences@tu-ilmenau.de)

ISBN: 978-3-938843-53-6 (USB-Flash Version)

Online-Version:

Publisher: Universitätsbibliothek Ilmenau
[ilmedia](#)
Postfach 10 05 65
98684 Ilmenau

© Ilmenau University of Technology (Thür.) 2010

The content of the USB-Flash and online-documents are copyright protected by law.
Der Inhalt des USB-Flash und die Online-Dokumente sind urheberrechtlich geschützt.

Home / Index:

<http://www.db-thueringen.de/servlets/DocumentServlet?id=16739>

CREATING A DISTRIBUTED DEVELOPMENT ENVIRONMENT FOR UNMANNED AERIAL VEHICLES USING USARSIM

Sebastian Drews, Sven Lange and Peter Protzel

Chemnitz University of Technology,
Department of Electrical Engineering and Information Technology

ABSTRACT

In this paper we discuss our efforts to create a dedicated simulation environment around USARSim for our autonomous mobile sensor platform "skeye-Copter". The platform is based on a multirotor aerial vehicle - a so-called quadcopter. Our work includes the development of a transparent software interface between the simulation and already existing software modules which allows the transition of algorithms between both - the simulated and the real robot. Furthermore, we compare and evaluate the applicability of the robot simulation to design and develop image processing based controls. In particular, we validate an approach for landing and position control of an autonomous multirotor aerial vehicle using the simulation environment.

Index Terms— unmanned aerial vehicles, autonomous systems, simulation, USARSim

1. INTRODUCTION

Our research focuses on enabling autonomous, mobile systems for the usage in a variety of civil applications, mainly in the areas of emergency response, disaster control and environmental monitoring. These scenarios require a high level of autonomy, reliability and general robustness of the used robotic system.

In this respect the development and testing of autonomous processes executed on highly agile robots like unmanned aerial vehicles substantially increases the demand for sophisticated simulation environments. Apart from the reduction of accident hazards, a solid simulation allows studies to be performed in noncritical environments and enables the use of software prototyping methods.

2. THE QUADROPTER "HUMMINGBIRD"

As the basis for our quadrotor we use the *Hummingbird* [1] which is manufactured by the AscTec GmbH, Germany. Measuring about 50 cm in diameter the quadrotor weighs between 350 g without and 500 g including a power supply. The basic equipment consists of an accelerometer, three electronic gyroscopes, three

magnetic field sensors, one for each spatial dimension, as well as a pressure-sensor and a GPS module. The quadrotor provides a hard-wired Philips LPC2146 microcontroller containing a 60 MHz 32-bit ARM processor which controls the thrust of the four rotors. It includes a control to balance the quadrotor horizontally and to regulate its height using the installed sensors. Since the height control mainly uses combined sensor data from the pressure and GPS sensor it is unsuitable for indoor use. In this described configuration, which is shown in Fig. 1, the quadrotor is able to fly for about 20 minutes using the included 2100 mAh lithium polymer battery.

To enhance its capabilities we extended the *Hummingbird* quadrotor in several ways. We constructed a wooden frame mounted on the bottom of the quadrotor which increases the space for additional sensors and computer systems. Namely we added a Logitech USB camera, a ADNS 3080 optical sensor, several SRF10 ultrasonic rangefinders as well as a *Gumstix* verdex XL6P, a fully-featured single-board computer, two 8-bit *ATmega644P* microcontrollers and a XBee Pro RF module.

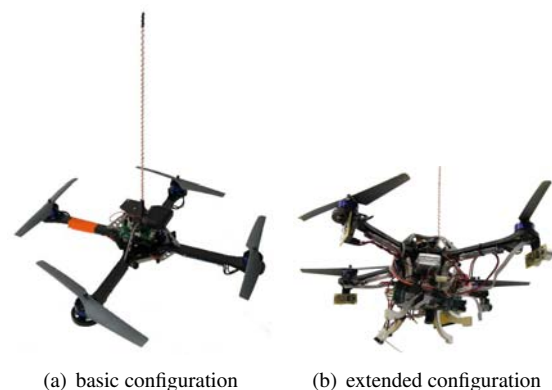


Fig. 1. The quadrotor *Hummingbird*

2.1. The Control Software

In the basic configuration the quadrotor can only be controlled manually using a classical radio control or by sending specific data packets to the hard-wired mi-

crocontroller over a serial connection. For this purpose the manufacturer specified a communication protocol providing 256 different packet types (PD). Since only a few standard packets are predefined the protocol can be extended according to the own requirements. The structure of such a data packet is shown in Fig. 2.

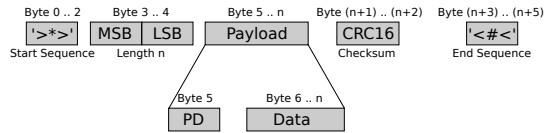


Fig. 2. Structure of the data packets

Its payload may contain commands as well as sensor data. The predefined packets only provide some basic flight commands and sensor data requests for the installed sensors. For this reason we developed a custom software interface that performs the high-level navigation control using the additional sensors, two more microcontrollers and the *Gumstix* computer. The underlying architecture of this interface is illustrated in Fig. 3.

The *ATmega644P_1* microcontroller acts as a proxy server between the hard-wired microcontroller, the *Gumstix* computer and the ground control. It is equipped with two serial ports and an I²C bus interface. The first serial port is connected to the hard-wired microcontroller. The second port is connected to a XBee Pro RF module used for wireless communication with remote clients. Generally this is the ground control - a graphical user interface running on a certain host. The I²C interface is used to communicate with the *Gumstix* computer as well as the second *ATmega644P* microcontroller and to read measured ranges from the SRF10 rangefinders. Whenever the *ATmega644P_1* microcontroller receives one of the predefined *Hummingbird* control packets from a remote client it simply forwards them to the hard-wired microcontroller and vice versa. Any other packet will be either processed by the *ATmega644P_1* itself or forwarded to the *Gumstix* computer. Basically the *ATmega644P_1* processes low-level motion control commands.

The *Gumstix* computer hosts a Linux distribution based on the OpenEmbedded Framework [2]. It contains the *skeyePilot*, a multi-threaded navigation control software application which has been implemented using Python. The *skeyePilot* includes the image processing which is implemented within a separate C++ library called *skeyeVision*. The graphical user interface at the ground control allows us to specify absolute velocities, distances relative to its current position as well as GPS coordinates. Furthermore, an autonomous landing system [3] can be enabled using the GUI. As can be seen in Fig. 3 an additional *ATmega644P* microcontroller is connected to the first one. It is equipped with the ADNS 3080 optical sensor and its primary task

is to perform optical flow motion estimation.

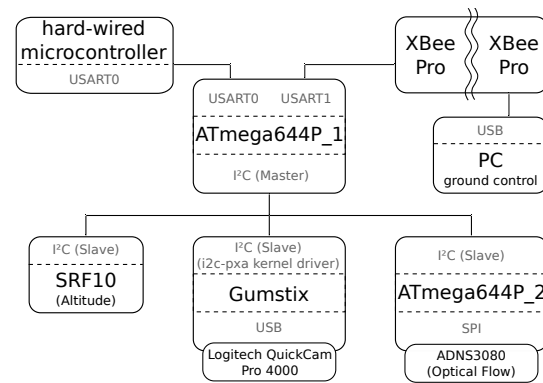


Fig. 3. Schematic diagram of the software components and communication channels on the quadrotor

3. AVAILABLE SIMULATION ENVIRONMENTS

There are several simulation environments and middleware software solutions available to simulate robots. Each of them provide a different degree of complexity and realism. The simulation environment must provide a 3D visualization or at least support the integration of an external graphics engine. A physics engine should be integrated or integrable as well. The creation of new robot models and the modification of existing robots regarding their technical data, physical properties and appearance should be straightforward. Beside the simulation of ground vehicles it has to support aerial vehicles as well. Furthermore, the simulation should enable us to mount cameras on the simulated robot and provide images captured from these cameras for image processing. It should be possible to remotely control the simulation platform. Preferably the control should be possible using any programming language but at least Python or C/C++ to ease the integration in the existing software architecture as described in the previous chapter.

Keeping the given requirements in mind three candidates remained which more or less fulfilled these criteria.

3.1. Player, Stage and Gazebo

The Player project [4], which originally was started by the Robotics Research Lab of the University of California, includes a 2D robot simulator called Stage and a 3D robot simulator called Gazebo as well as a distributed robot control interface called Player. Since the communication with the robot control is based on a TCP/IP network protocol it gives you the freedom to chose any programming language (supporting socket communication) and development environment to implement high-level applications. Besides the environ-

ment provides a C/C++ library and python bindings. Gazebo uses the Open Dynamics Engine¹ for physics and the Object-Oriented Graphics Rendering Engine². The modeling of robots, sensors and worlds is done using XML configuration files. A graphical editor does not exist which makes the construction of complex worlds quite difficult and time-consuming. It seems that Gazebo is not fully developed yet. It is not as stable as desired and the documentation is not up to date.

3.2. Webots

Another solution that meets most of our requirements was the commercial robot simulation Webots [5] developed by the swiss company Cyberbotics. It started once back in 1996 as a project of the École Polytechnique Fédérale de Lausanne (EPFL). There exist versions for Windows, Linux and Mac OS. The simulation also uses ODE as physics engine. VRML97 gets used to store the descriptions of the worlds which allows the usage of several available modeling tools. It is a locally executed application not a distributed system. Simulation, control and visualization are combined within the graphical user interface. The source code of the simulation is not available. Prices for the education edition of the software start at about 200€. Webots simulates all the conventional sensors. Besides cameras, light, touch, pressure and GPS sensors even gyroscopes and acceleration sensors are included. The simulation emulates several ground vehicles as well as an airship. For some platforms it even offers cross-compiling support. Due to its commercial character there is a lack of freely available resources like worlds and robot models. Since it is a closed source application one can not comprehend how a certain robot or sensor model works. Furthermore, it is not possible to create custom sensors. In our opinion the named disadvantages of this solution outweigh its features.

3.3. USARSim

USARSim - *the Unified System for Automation and Robot Simulation* [6] is an open-source high-fidelity simulation of robots, sensors and environments based on the popular middleware Unreal Engine which is used in many commercial computer games. Since its introduction in 2002 it has been evolved from a plain solution for search and rescue scenarios into a versatile research platform for autonomous robots. The simulation is used in the yearly RoboCup Rescue Simulation League as well as the Virtual Manufacturing Automation Competition which was held in conjunction with the IEEE International Conference on Robotics and Automation (ICRA) 2010. Version 2.5 of the Unreal Engine consists of the Karma physics engine and a

high-end graphics engine. Since version 3.0 Karma has been replaced by Nvidia's PhysX engine. To use the USARSim simulation environment it is required to install the commercial game Unreal Tournament 2004 or Unreal Tournament 3 for the new version. Both games can be purchased for less than 10€. In November 2009 Epic Games released the *Unreal Development Kit* including a free edition of the Unreal Engine 3.0. Alternatively to the commercial games a slightly modified version of USARSim can be used with the UDK instead. Since the most recent version of the simulation which is based on Unreal Engine 3 is still under heavy development, we used the final version of USARSim 2004 which is based on the Unreal Engine 2.5.

The Unreal Engine offers an object oriented programming language called Unreal Script which allows the modification and extension of the game. USARSim is completely written in Unreal Script. The communication interface between the simulation and the controller is realized using a modification of Gamebots, an open-source client-server solution for the Unreal Engine. That way the controller can send commands to the simulated robot and receive sensor data from it. The architecture is shown in Fig. 4.

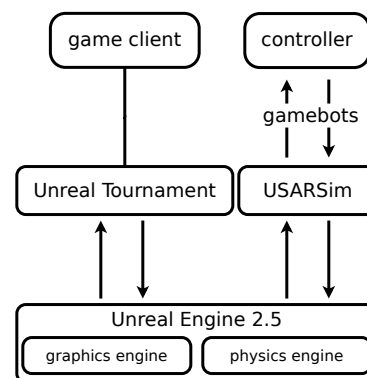


Fig. 4. Architecture of USARSim

An Unreal Engine based application is executed in a so-called Unreal Virtual Machine. The UVM consists of the Unreal server, a client, the graphics- and physics engine and several management tasks. Server and client may run on the same system. The state of each actor within the engine is updated after a certain period of time called "Tick". The length of that period of time directly depends on the available computational power. Typical values are low hundredths of a second. The tick rate of the server is equivalent to the frame rate on the client side. To parallelize the execution of different actors the Unreal Engine internally simulates threads. Each actor gets executed within its own thread. After each tick period several event handler and actor specific functions get called. They process commands and generate responses to environmental influences.

Unreal Tournament includes a graphical editor to create and modify the shape and basic physical prop-

¹<http://www.ode.org>

²<http://www.ogre3d.org>

erties of worlds, actors, sensors and other objects. The skin of an object can be created using a 3D graphics application like Blender, Lightwave or 3D Studio Max.

The details of the physical characteristics of an object are specified using customizable attributes of the physics engine. Within a configuration file one can construct certain robots, specify which sensors get mounted on a robot including their position, orientation and basic attributes like maximal ranges etc. Currently the simulation environment contains more than 30 different robots and sensor models including the AirRobot quadrotor which we decided to use for our purpose.

3.4. Summary

We have to say that none of the evaluated robot simulations provides a physical model which allows us the emulation of a real atmosphere and thus the simulation of real flight principles.

The main criteria that lead to our decision in favor of USARSim was that it already includes several Urban Search and Rescue scenarios and the fact that it provides all the robot and sensor models we need. The quality of the graphics engine is outstanding and the physics engine also performs well. Using the included graphical editor it is relatively simple to create, modify or extend the simulated environment. Furthermore, USARSim is open source and the complexity of the underlying Unreal Script code is also manageable. The accuracy and degree of realism of the simulation have been successfully validated already in several analyses like for instance [7] or [8].

4. THE CONTROL INTERFACE

Using USARSim our goal was to create a transparent control interface that enables us to execute the existing control applications on the real robot or in the simulated world as desired without changing the underlying program code. We could benefit from the fact that both the communication with USARSim and with the quadrotor were based on a network protocol. The challenge was to connect both interfaces and to emulate certain hardware components within different software modules. An overview of our solution can be seen in Fig. 5.

The first step was to create a Python class structure to reflect the robot and sensor models provided by USARSim. We created classes for each sensor and robot subdividing similar models into subclasses of a base class.

After initializing a robot within the simulation it continuously sends informations about the robots state and sensor data to the controller at a predefined frequency. This is the reason for the need of a dedicated instance called message handler that processes the received data and updates the corresponding sensor ob-

ject attributes. On the other side a dedicated command handler is needed since sending commands must also be exclusive to preserve a consistent state of the robot.

The third component of the control interface is the motion control. It includes several proportional-integral controllers for simple linear-, lateral- and angular movement as well as coordinated navigation using GPS or world coordinates. The communication between the concurrent processes is solved using multi-producer, single-consumer queues. They have a limited size to prevent flooding by the producer. The described four components are used within a robot specific supervisory class which provides several methods to control the simulated robot and to read sensor data. This way we developed a Python application programming interface that allows the control of a robot with only a few lines of code.

The transition between the existing software and the simulation has been realized within a separate process. Therefore, we extended the communication interface of the client software to optionally use a TCP/IP network connection besides the existing serial line connection. The so-called *simPilot* combines the functionality of the *ATmega644P_1* microcontroller, the hardwired microcontroller as well as the *Gumstix* computer. It listens on a network socket for an incoming connection while only one client is supported. The *simPilot* utilizes the existing modules for packet encoding and decoding which are part of the ground control software. In contrast to the *ATmega644P_1* microcontroller on the real robot it does not forward any packet but processes all of them directly. The client can be an arbitrary application which just implements the described communication protocol. In our case the graphical user interface of the ground control can now be used to control a robot within the simulated world. The only change that has to be done is to enable the network support within its configuration file and to specify the host and port of the *simPilot*.

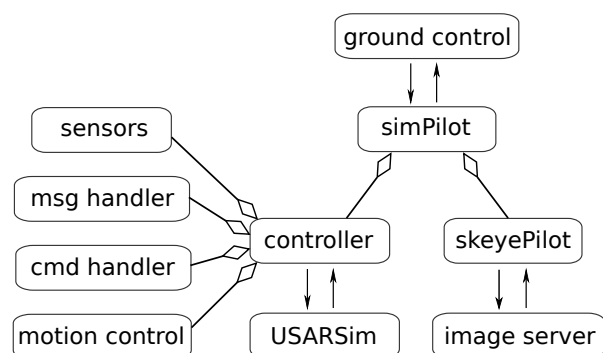


Fig. 5. Architecture of the simulation environment

Unreal Script does not provide functions to export pictures from within the simulated world. For this reason we had to find an external solution which is not that easy since the Unreal Engine itself is closed-source.

The trivial approach to solve this problem is to simply capture screenshots of the running application. Using Linux as development environment we evaluated several ways to capture a screenshot of a X11 application using basic X11 functions and the Composite Extension. None of them delivered the desired performance to emulate a real camera with a frame rate of about $30Hz$. The Linux version of Unreal Tournament uses the Simple Directmedia Layer Library [9] to abstract certain low-level input- and output-functions. This fact provided a different approach for the solution since the libSDL also encapsulates the underlying OpenGL functions. This enabled us to read the content of the OpenGL back buffer and thus retrieve the current image frame just before it got rendered to the screen. We extended the SDL library³ to include a multi-threaded image server providing both RAW and compressed images using the Portable Network Graphics library [10] or the JPEG library [11]. Using this solution we achieved a frame rate of about $30Hz$ for JPEG compressed images and $60Hz$ for RAW images at a resolution of 800×600 pixels while the Unreal Engine, USARSim and the *simPilot* were running. Which results in $5 + n$ threads where n is the current number of clients of the image server. The measurements have been made on a computer system equipped with an Intel Core 2 Quad Q9550 and a Nvidia GeForce 9500 GT.

5. VALIDATION

To validate the applicability of the developed interface we compared the results of a high-level image processing method running on the real robot and within the simulation. The chosen task was a landing pad detection using a camera and a custom target pattern [3]. The pattern consists of several concentric white rings on a black background. Each of the white rings has an unique ratio of its inner to outer border radius. Therefore, the rings can be uniquely identified. Using this pattern the method calculates the distance between the current position of the robot and the landing pad as well as the current height of the robot. The height h can be calculated as

$$h = \frac{r_i[cm]}{r_i[pix]} \cdot f \quad (1)$$

where f is the focal length, $r_i[cm]$ is the real radius of one of the white rings in centimeter and $r_i[pix]$ is the radius of the ring on the camera image in pixel.

During this validation process we could observe deviations between the calculated height and the real height provided by the ground truth of up to 5%. Although the percentage is relatively low the different heights are not a result of errors produced by the calculation method. The cause for the deviation is the

³The extension is available at: <http://www.tu-chemnitz.de/etit/proaut/forschung/simulation.html.en>

simulation itself. To display a certain field of view the method `Canvas.DrawPortal()` gets used. This method expects the location, orientation and the field of view as parameters. The FOV is given in degrees as an integer value. Floating point values are always rounded down resulting in a maximal inaccuracy of 1 degree. Keeping this fact in mind and using only integer values for the FOV we could reduce the deviation to 2%. To validate the landing pad detection we added a mock-up of the landing pad to the used map and mounted a camera with a FOV of $1.309rad$ (75 degrees) on the robot. Starting at a height of $30cm$ directly above the landing pad we gradually increased its height with a maximal speed of $0.1 \frac{m}{s}$ up to $1.0m$. After staying at this height for $10s$ the robot flew at the height of $2.0m$ staying there for another $10s$.

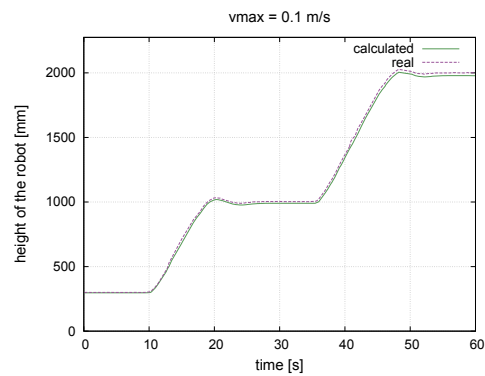


Fig. 6. Comparison of calculated and real height

As can be seen in Fig. 6, again there is a deviation between the results of the measurement and the calculation. This time its maximum value reaches almost up to $2cm$. In this case the reason is that the size of the displayed landing pad can not be converted exactly into pixels. There will always be inaccuracies and rounding errors. The overshoots are caused by latencies of the interprocess communication between the different instances and possible timeouts within the control application.

Afterwards we compared the behavior of the landing pad detection in both scenarios. We emulated atmospheric influences within the simulation through applying a time-varying force to the robots center of mass. The direction and absolute value of the force vector were chosen randomly for each tick period. The robot's start position was $70cm$ above the landing pad. The maximal linear and lateral velocities of the motion control were limited to $0.3 \frac{m}{s}$. The maximal difference between the current and the target position was specified with $5cm$. Only when the difference was above this limit the motion control became active. The results of these measurements can be found in Fig. 7.

In both scenarios the robot flew 5 minutes above the landing pad. Without the landing pad detection this would result in a continuously drifting robot. With ac-

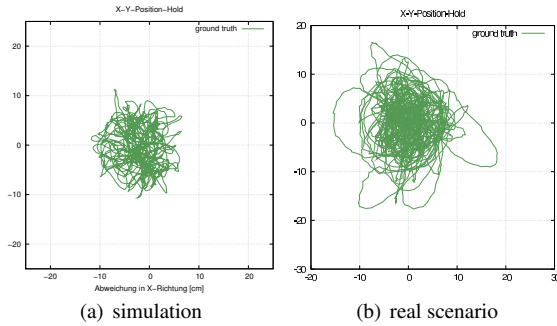


Fig. 7. Distance between the centre of the robot and the centre of the landing pad with activated landingpad detection

tivated landing pad detection the quadrotor kept its position. While the standard deviation (3.8cm to 2.49cm) and maximum (23cm to 13.27cm) of the position offset are larger in the real scenario than in the simulation the effects of the algorithm are comparable. However, the results of changes of the algorithm are expected to be similar in the simulation and the real world. Thus the further development of the landing pad detection could be done using the simulation.

6. RESULTS AND CONCLUSIONS

With the development of the *simPilot* and the underlying controller we realized the transparent software interface between USARSim and our ground control application that we aimed at. Unfortunately the Unreal Engine does not provide a way to emulate real lift. As a result the aerial vehicles are based on comparatively simple models which hardly reflect the high sensitivity of the real robots. So as expected the simulation does not enable us to develop low-level control processes e.g. for navigation purposes that are applicable on the real vehicle. But it is a valuable tool to create and test new high-level algorithms based on different sensors and image processing. Such processes can be adopted almost directly from the real world and vice versa since the optical attributes and sensor properties are equivalent and only certain parameters need to get adjusted as we could demonstrate with the landing pad detection. In a recent project we used the simulation environment to validate a vision based approach for SLAM in indoor environments [12].

Generally the simulation environment allows us to efficiently evaluate whether new strategies to solve certain problems are worth further investigation. Future work will be dedicated to the integration of our ground vehicles including the corresponding modifications and enhancements of the controller. Another task will be the modeling of existing real world environments as Unreal maps and to extend the software interface to use it with the Unreal Engine 3.0.

7. REFERENCES

- [1] Daniel Gurdan, Jan Stumpf, Michael Achtelik, Klaus-Michael Doth, Gerd Hirzinger, and Daniela Rus, “Energy-efficient Autonomous Four-rotor Flying Robot Controlled at 1 kHz”, in *Proc. of IEEE International Conference on Robotics and Automation, ICRA07*, 2007.
- [2] “OpenEmbedded Framework”, <http://www.openembedded.org>, 2010.
- [3] Sven Lange, Niko Sünderhauf, and Peter Protzel, “A Vision Based Onboard Approach for Landing and Position Control of an Autonomous Multicopter UAV in GPS-Denied Environments”, in *Proc. of the International Conference on Advanced Robotics, ICAR09*, 2009.
- [4] Player, “Player/Stage/Gazebo, version 3.0/0.9”, <http://playerstage.sourceforge.net>, 2009.
- [5] Webots, “Webots, version 6.1.5”, <http://www.cyberbotics.com>, 2009.
- [6] USARSim, “Unified System for Automation and Robot Simulation”, <http://sourceforge.net/projects/usarsim/>, 2010.
- [7] S. Hughes M. Koes J. Wang, M. Lewis and S. Carpin, “Validating USARsim for use in HRI Research”, in *Proc. of the Human Factors and Ergonomics Society 49th Annual Meeting*, 2005, pp. 457–461.
- [8] S. Carpin, T. Stoyanov, and Y. Nevatia, “Quantitative assessments of USARSim accuracy”, in *Proc. of PerMIS 2006*, 2006.
- [9] libSDL, “Simple Directmedia Layer, Version 1.2.14”, <http://www.libsdl.org>, 2009.
- [10] libpng, “Portable Network Graphics Library, Version 1.2.37”, <http://www.libpng.org>, 2009.
- [11] libjpeg, “Independent JPEG Group’s free JPEG software: LIBJPEG, Version 6b”, <http://www.ijg.org>, 2009.
- [12] Sven Lange and Peter Protzel, “Active Stereo Vision for Autonomous Multicopter UAVs in Indoor Environments”, in *Proc. of the 11th Conference Towards Autonomous Robotic Systems, TAROS 2010*, 2010.